The “why did you do that?” button

Koeman, Vincent J.; Dennis, Louise A.; Webster, Matt; Fisher, Michael; Hindriks, Koen

published in
Engineering Multi-Agent System
2020

DOI (link to publisher)
10.1007/978-3-030-51417-4_8

document version
Publisher's PDF, also known as Version of record

The "Why Did You Do That?" Button: Answering Why-Questions for End Users of Robotic Systems

Vincent J. Koeman[1], Louise A. Dennis[2(✉)], Matt Webster[2], Michael Fisher[2], and Koen Hindriks[3]

[1] Delft University of Technology, Delft, The Netherlands
v.j.koeman@tudelft.nl
[2] University of Liverpool, Liverpool, UK
{L.A.Dennis,M.Webster,MFisher}@liverpool.ac.uk
[3] Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
k.v.hindriks@vu.nl

Abstract. The issue of explainability for autonomous systems is becoming increasingly prominent. Several researchers and organisations have advocated the provision of a "Why did you do that?" button which allows a user to interrogate a robot about its choices and actions. We take previous work on debugging cognitive agent programs and apply it to the question of supplying explanations to end users in the form of answers to *why-questions*. These previous approaches are based on the generation of a trace of events in the execution of the program and then answering why-questions using the trace. We implemented this framework in the *agent infrastructure layer* and, in particular, the GWENDOLEN programming language it supports – extending it in the process to handle the generation of applicable plans and multiple intentions. In order to make the answers to why-questions comprehensible to end users we advocate a two step process in which first a representation of an explanation is created and this is subsequently converted into natural language in a way which abstracts away from some events in the trace and employs application specific *predicate dictionaries* in order to translate the first-order logic presentation of concepts within the cognitive agent program in natural language. A prototype implementation of these ideas is provided.

1 Introduction

As autonomous systems become more prevalent in society, issues related to the ways in which humans interact with such systems become more important. Among these issues is the question of transparency and, in particular, explainability. Wortham and Theodorou [35], and Sheh [24] (among others) have argued that the ability for a robot (and by extension any autonomous system) to provide explanations of its behaviour helps users develop accurate mental models of the robot's reasoning and so interact better with the robot and develop trust. Charisi

L. A. Dennis et al. (Eds.): EMAS 2019, LNAI 12058, pp. 152–172, 2020.
https://doi.org/10.1007/978-3-030-51417-4_8

et al. [5], Turner [26] and The IEEE Global Initiative on Ethics of Autonomous and Intelligent Systems [25] in particular advocate the provision of a "why did you do that?" button to help the user understand a robot's behaviour.

We take as our focus autonomous systems which employ a cognitive agent to make high level decisions such as [27,28,36]. One of the reasons often put forward for the employment of cognitive agents in this role is their in principle ability to explain their decisions to end users. However, in practice, little research has been performed in actually providing such explanations of reasoning.

There are a number of key problems in the provision of explanations. Firstly, they require a backward view of the program execution (in contrast to common debugging practice in which a breakpoint is set and the program is then run forwards from the breakpoint). Secondly, log files, which are the obvious solution to the first problem tend to be verbose and their production can cause significant performance overheads. These problems are exacerbated when all the information needed to understand why something is taking place must be captured.

In this paper we combine work on the debugging of cognitive agent programs in the Beliefs-Desires-Intentions (BDI) paradigm [17] with work on the provision of explanations for programmers in GOAL [15] and AgentSpeak [31]. Koeman et al. [17] generate an omniscient trace of key events that take place during program execution in a manner which limits the overhead cost of producing the trace. Each event stores enough information about the agent's mental state to reconstruct the state of the program execution at that point. This trace is supported by tools allowing it to be viewed at a high-level of abstraction hiding extraneous information unless a user wants to see it.

We have implemented omniscient tracing in the *Agent Infrastructure Layer* (AIL) [7,9], a prototyping tool for verifiable interpreters for cognitive agent programming languages, with particular attention to the Gwendolen programming language [8] but with attention paid to keeping the framework generic where possible. In applying this framework to the AIL we extended the key events considered beyond changes to the agent's mental state to include a number of events involved in the generation of plans and the handling of intentions.

The development of omniscient debugging was driven, in part, by a desire to support programmers in answering why-questions. Programmers can interrogate the high level trace at specific points in the program execution and ask "why did you do that" (as outlined in [15]). Winikoff [31] reports on a similar system constructing why and why-not explanations over traces for AgentSpeak.

We implemented this idea in our AIL-based omniscient debugging framework. We developed an explanation generation framework for end users that is specific to Gwendolen, providing explanations at a higher level of abstraction than previously considered, and using *predicate dictionaries* to provide natural language substitutes for application specific logical predicates. This implementation generates explanations when multiple intentions are being executed in an interleaved fashion (something omitted from [31]).

2 Background and Related Work

2.1 Cognitive Agent Programming

At its most general, an *agent* is an abstract concept that represents an *autonomous* computational entity that makes its own decisions [33]. A general agent is simply the encapsulation of some distributed computational component within a larger system. However, in many settings, something more is needed. Rather than just having a system which makes its own decisions in an opaque way, it is increasingly important for the agent to have explicit *reasons* (that it could explain, if necessary) for making one choice over another.

Cognitive agents [3,21,34] enable the representation of this kind of reasoning. Such an agent has explicit reasons for making the choices it does. We often describe a cognitive agent's *beliefs* and *goals*, which in turn determine the agent's *intentions*. Such agents make decisions about what action to perform, given their current beliefs, goals and intentions. This view of cognitive agents is encapsulated within the Beliefs-Desires-Intentions (BDI) model [20–22]. Beliefs represent the agent's (possibly incomplete, possibly incorrect) information about itself, other agents, and its environment, desires represent the agent's long-term goals while intentions represent the goals that the agent is actively pursuing (the representation of intentions often includes partially instantiated and/or executed plans and so combines the goal with its intended means).

There are *many* different agent programming languages and agent platforms based, at least in part, on the BDI approach [1,6,14,19,23]. Agents programmed in these languages commonly contain a set of *beliefs*, a set of *goals*, and a set of *plans*. Plans determine how an agent acts based on its beliefs and goals and form the basis for *practical reasoning* (i.e., reasoning about actions) in such agents. As a result of executing a plan, the beliefs and goals of an agent may change and actions may be executed.

It is generally recognised that debugging BDI agent programs is hard [29,30] (and by extension that agent behaviour can be difficult to understand even when performing as desired). In particular agents react to exogenous events in dynamic environments; exogenous events which may combine in unexpected ways and which may be handled by the agent "in parallel" with each other. Furthermore many cognitive agent languages have provision for failure handling which, again, may interact in complex ways with the behaviour of the rest of the program.

2.2 Explanations in Cognitive Agent Systems and "Why" Questions

Ko and Myers [16] created the WHY-LINE tool, which allows developers to pose "why did" or "why didn't" questions about the output of Java programs. A trace is generated in memory through bytecode instrumentation, containing everything necessary for reproducing a specific execution. From this trace, a set of questions and associated answers is generated. The authors note that their approach is not suited for executions that span more than a few minutes or executions that process or produce substantial amounts of data. However,

their results do show that the approach enables developers to debug failures substantially faster.

Hindriks [15] and Winikoff [31] both consider a similar model applied to the debugging of cognitive agent programs in GOAL and AgentSpeak respectively. Of these [15] is the earlier and has a more informal treatment than [31] which sought to extend, formalise and implement the proposal. The two approaches are therefore similar in their underlying conception and we use them as the basis for our work. The key idea is that a trace of events is stored as a log. Each event in the trace can be interrogated and an explanation constructed in a systematic way using information either stored in that event and/or by referring to a previous event in the log. For instance the explanation for why some action was executed might be that "the action's preconditions held and a plan was previously selected which contained the action". Explanations can then also be given for why the preconditions succeeded and/or why the plan was selected.

Koeman et al. [17] propose a trace based mechanism for debugging cognitive agent programs. Although concerned with many of the same issues as [15] and [31] (and indeed, intended as support for the mechanisms proposed in [15]) the authors focus on more foundational questions of what information needs to be stored in a trace in order to reconstruct the state of an agent at that point, and the performance overhead of storing such traces for a program. They conclude that if a trace stored the key events in agent execution, namely the changes to the agent's mental state, then the program run could be reconstructed without the significant performance impact associated with storing the full state of an agent at each step in execution. They develop a *space-time visualiser* for these traces which allows a programmer to inspect the trace and query the state of the underlying program at any point.

Hindriks [15] and Koeman et al. [17] consider primarily changes to an agent's mental state (i.e., beliefs and goals) in their tracing and debugging frameworks. Winikoff [31] extends this to include traces and explanations for the selection of plans but assumes that the entirety of a plan is executed before anything else happens. The AIL allows interleaved execution of plans by manipulating intentions. In our work therefore, we integrated the approach in [31] with that of [17] and then extended it to the handling of multiple intentions[1].

A few systems have considered the question of providing explanations specifically for end users of cognitive agent systems. In Harbers [13] explanations of agent behaviour are generated based on the beliefs and goals of the agents using a goal hierarchy paired with a behaviour log. Winikoff et al. [32] presents a similar system but adds the concept of preferences (or *valuings*) to the explanations presented to end users. The use of goal heirarchies can be viewed as a more abstract approach than ours which considers the concept of plans and their selection as an important part of explanation generation beyond their use

[1] Though it should be noted that the implementation of omniscient debugging in GOAL also handles GOAL's module mechanism (although this is not reported in depth in [17]) which is not entirely dissimilar to the concept of intention in the AIL.

to decompose goals into sub-goals. We hypothesise that many users will find the concept of plan a useful one but have yet to evaluate this hypothesis in any way.

2.3 The Agent Infrastructure Layer and Gwendolen

The Agent Infrastructure Layer (AIL) [7] is a set of Java classes intended to assist in the development of BDI-style programming languages. Gwendolen [8] is the most mature language in this framework.

Aside: It is unfortunate that the literature on tracing programs refers to storing key events in a trace, while the BDI literature refers to events that trigger plans (which may be either external or internal to the program). We distinguish between these two uses of the word "event" in what follows by using *trace event* for events stored in traces and *BDI event* for events that may trigger plans during program execution.

The AIL provides default data structures for agents, beliefs, goals, plans and intentions. Individual languages implemented in the AIL define custom reasoning cycles for agent deliberation. However the toolkit has an underlying assumption that such reasoning cycles will typically involve the following steps in some order:

- Perception which creates sets of new beliefs and removes beliefs that no longer hold.
- Posting BDI events (either as new intentions, or added to existing intentions) when beliefs are acquired or removed and goals are acquired or removed.
- Selecting plans to react to BDI events.
- Selecting among intentions which represent partially processed plans or unhandled BDI events.
- Processing one (or more) steps in an intention which include adding and removing beliefs and goals and executing actions.

These default steps therefore form the core events supported by our implementation of omniscient debugging within the AIL.

Gwendolen Operational Semantics. We use Gwendolen as our key implementation language. We present here a simplified version of the Gwendolen operational semantics which is presented in full in [8]. The semantics presented here assumes all terms are ground (so ignores issues surrounding the handling of unifiers), and ignores a number of language features such as locking and suspending intentions, dropping goals, agent sleeping and waking behaviour, message handling and special cases such as transitions for handling goals that can't be planned. The intention is to present enough information to allow our framework to be understood. This operational semantics is shown in Fig. 1. Following [31] we annotate the transitions (expressions above the arrow) with the trace events that are stored by the omniscient debugger. These are discussed further in Sect. 3.

$$\frac{\mathcal{S}_{\text{int}}(I \cup \{i_k\}) = (i'_{k'}, I')}{\langle i_k, I, B, \emptyset\rangle \xrightarrow{sel i(i'_{k'})} \langle i'_{k'}, I', B, \emptyset\rangle}\mathbf{A} \quad (1)$$

$$\frac{\mathcal{G}(i_k, I, B) = A}{\langle i_k, I, B, \emptyset\rangle \xrightarrow{\Gamma} \langle i_k, I, B, A\rangle}\mathbf{B} \quad (2)$$

$$\frac{(e, ds) = \mathcal{S}_{\text{plan}}(A)}{\langle i_k, I, B, A\rangle \xrightarrow{selp((e,ds),k)} \langle (e, ds) \mathbin{@} \mathtt{tl}_i(i_k), I, B, \emptyset\rangle}\mathbf{C} \quad (3)$$

$$\frac{B \models g}{\langle (e, +!g);_i i_k, I, B, \emptyset\rangle \to \langle i_k, I, B, \emptyset\rangle}\mathbf{D} \quad (4)$$

$$\frac{B \not\models g}{\langle (e, +!g);_i i_k, I, B, \emptyset\rangle \xrightarrow{add((e,+!g),k)} \langle (+!g, \epsilon);_i (e, +!g);_i i_k, I, B, \emptyset\rangle}\mathbf{D} \quad (5)$$

$$\frac{}{\langle (e, +b);_i i_k, I, B, \emptyset\rangle \xrightarrow{addb(+b,k);crei((+b,\epsilon),k')} \langle i_k, I \cup (+b, \epsilon)_{k'}, B \cup \{b\}, \emptyset\rangle}\mathbf{D} \quad (6)$$

$$\frac{}{\langle (e, -b);_i i_k, I, B, \emptyset \ldots\rangle \xrightarrow{delb(-b,k);crei((-b,\epsilon),k')} \langle i_k, I \cup (-b, \epsilon)_{k'}, B \backslash \{b\}, \emptyset\rangle}\mathbf{D} \quad (7)$$

$$\frac{\mathbf{do}(a)}{\langle (e, a);_i i_k, I, B, \emptyset\rangle \xrightarrow{act(a,k)} \langle i_k, I, B, \emptyset\rangle}\mathbf{D} \quad (8)$$

$$\frac{P = \mathbf{Percepts} \quad OP = \{b \mid b \in B \backslash P \wedge percept(b)\}}{\langle i_k, I, A, B\rangle \xrightarrow{\Pi} \langle i, I \cup \{(\mathtt{percept}, +b)_{k'} \mid b \in P \backslash B \wedge fresh(k')\} \cup \{(\mathtt{percept}, -b)_{k'} \mid b \in OP \wedge fresh(k')\}, B, A\rangle}\mathbf{E} \quad (9)$$

Fig. 1. Simplified Gwendolen Semantics

The Gwendolen reasoning cycle shown here has five stages **A**, **B**, **C**, **D** and **E**[2]. One transition in each stage is executed in turn. In the semantics we show the stage that a transition applies to with a letter to the right of the rule. A Gwendolen agent starts in stage **A** and so (1) is the first rule to apply, followed by (2) and so on. In stage **D** whichever rule applies to the top of the current intention is applied and then the reasoning cycle moves on to stage **E**.

BDI languages use intentions to store the *intended means* for achieving goals – this is generally represented as some form of *deed stack* (deeds include actions, belief updates, and the commitment to goals). In Gwendolen, intention structures[3] also maintain information about the BDI event that triggered them (the addition or removal of a belief or the posting of a (sub-)goal). Gwendolen aggregates this information: an intention becomes a stack of tuples of an event and a deed. Each tuple associates a particular deed with the BDI event that triggered the plan that placed the deed in the intention. Unplanned BDI events are associated with an empty deed, ϵ, which can be thought of a marker indicating "no plan yet".

[2] The implementation of Gwendolen contains a sixth stage for message handling.
[3] A refinement of the AIL's intention structure which is more general.

In order to track the evolution of intentions in traces more easily, we extended the AIL implementation of intentions with an ID number, k, and will use the notation i_k to represent that intention, i, has ID number, k. This ID number is frequently stored in trace events (see, for instance, (3) in Fig. 1).

We represent an agent state as a tuple $\langle i, I, B, A\rangle$ where: i is the current intention; I is a queue of intentions $\{i_1, i_2, ..\}$; B is a set of the agent's beliefs; and A is a set of currently applicable plans for the current intention i.

A Gwendolen program consists of a set of plans, Δ, of the form, $e : \{g\} \leftarrow ds$ (where ds is a sequence of deeds to be executed if BDI event, e is posted and guard, g, follows from the agent's beliefs and goals), a set of initial beliefs, $\mathcal{B}$, and a set of initial goals, Gs. In an agent's initial state the current intention is *null*, the intention set consists of one intention for each of the initial goals provided by the programmer of the form $(\texttt{start}, +!g)$. The belief base is $\mathcal{B}$ and the applicable plans are empty.

(1) governs the selection of intentions. $\mathcal{S}_{\text{int}}$ is an application specific function that selects one intention out of a set of intentions and returns a tuple of the selected intention and the set without that intention in it. By default $\mathcal{S}_{\text{int}}$ operates on a queue data structure and so in general the current intention is placed at the end of the queue and the intention at the top of the queue is selected. Also by default empty intentions which have been fully executed are removed at this point.

(2) represents the process of inspecting the plan library and finding plans that match the current intention. These are transformed into *applicable plans* and returned by the function $\mathcal{G}$. A plan, $e : \{g\} \leftarrow ds$ matches an intention if e matches the BDI event in the top tuple of the intention, g is a logical consequence of the agent's beliefs and goals (goals are inferred from the BDI events posted in all intentions) and the deed in the top tuple of the intention is ϵ. Applicable plans are an interim data structure that describe how the plan changes the current intention. An applicable plan describes new tuples to be placed on the top of the intention stack (replacing the existing top tuple). A tuple is created for each deed in ds and associated with e[4].

(3) uses the application specific function $\mathcal{S}_{\text{plan}}$ to pick an applicable plan to be applied. By default, this treats the set as a list and picks the first plan based on the order they appear in the Gwendolen program. We use the syntax (e, ds) @ $\texttt{tl}_i(i)$ to represent the replacement of the top tuple in intention, i, by the tuples in the applicable plan, $e : \{g\} \leftarrow ds$.

(4), (5), (6), (7) and (8) process the top deed in the intention handling the instruction to add a goal (depending upon whether the goal is already achieved or not), add a belief, drop a belief and execute an action respectively. $(e, d);_i i$

[4] In order to handle situations where the top deed on the intention is not ϵ ("no plan yet") then $\mathcal{G}$ returns the existing top tuple so there is no change to the intention and it continues to be processed as normal. This somewhat baroque mechanism has its roots in Gwendolen's origin as an intermediate language into which all BDI languages could be translated [10]. We ignore this type of applicable plan in our explanation mechanism and so do not refer to them further here.

represents the addition of the tuple (e, d) to the top of the intention i. **do**(a) represents the execution of an action in some external environment. These rules make a check on the top deed in the intention to see what type it is (e.g., the addition of a belief, the deletion of a goal). We represent these checks implicitly using the notation: a for an action; $+b$ for a belief addition; $-b$ for a belief removal; and $+!g$ for a goal addition. (6) and (7) both add two trace events to the trace representing that both the belief base has been changed and that a new intention has been created. This new intention has a new ID number k'.

(9) handles perception. A set of **Percepts** are gathered from the environment. New percepts are added as intentions to add a belief (each with a new ID number indicated by $fresh(k')$). Out of date percepts (i.e., percepts in the belief base that can no longer be perceived) are handled by creating a new intention to remove them.

3 An AIL-Based Framework for Omniscient Debugging Driven Explanations for Cognitive Agents

As noted above, omniscient debugging was developed with the intention of supporting explanations in the form of answering why- and why-not-questions as outlined in [15] and [31].

Omniscient debugging for GOAL focused on the changes in agent goals and beliefs as the key trace events underpinning a trace. We used the analysis from Sect. 2.3 to extend[5] this to:

1. Creation of intention, i_k: $crei(i, k)$.
2. Selection of intention, i_k: $seli(k)$.
3. Successful evaluation of guard, g for (applicable) plan π, with unifier, θ in intention, i_k: $bel(\pi, g, \theta, k)$.
4. Selection of an applicable plan, (e, ds) in intention, i_k: $selp((e, ds), k)$.
5. Execution of action, a, by intention i_k: $act(a, k)$.
6. Adding or removing goal, g, by intention i_k: $addg(g, k), delg(g, k)$.
7. Adding or removing belief, b by intention i_k: $addb(b, k), delb(b, k)$.
8. Modification of intention, i_k, by adding or removing tuples, ts: $add(ts, k)$, $del(ts, k)$.

We used the work of Koeman et al. [17] for trace construction and visualisation to implement tracing in the AIL. Since both GOAL and the AIL were implemented in Java it was possible to port much of the framework directly.

3.1 Adaptation to Gwendolen

Commands to log these trace events were embedded in relevant parts of the AIL toolkit, primarily in classes used to implement transition rules in reasoning

[5] Note this is not the complete set of trace events shown in Fig. 1. This is elaborated further in Sect. 3.1.

cycles. This is why in Fig. 1 we were able to annotate the transitions in the Gwendolen semantics with the associated trace events. Given Gwendolen modifies the current intention when a goal is posted instead of maintaining a goal base we do not use $addg(g, k)$ or $delg(g, k)$.

In Fig. 1 we reference two further constructs, Γ and Π, these represent situations where one transition in the semantics generates several trace events in the trace. Γ logs each successful guard evaluation for plans in Δ as an event in the trace and associates them with the relevant applicable plan. Π logs the creation of the intentions caused by the addition and removal of beliefs following perception.

3.2 Example

As an example of a simple Gwendolen trace we consider the excution of a Gwendolen program that consists of a single plan $+b : \{a\} \leftarrow +d; e$ (if the BDI event that b is believed is posted and a is already believed then add the belief d and do e). We will omit the gory details of the Gwendolen agent state, but hope the process of execution is nevertheless comprehensible from the example trace.

In the trace a number of beliefs are added following perception steps in the program execution. Some of these beliefs are relevant to the plan execution and some are not. We have included them to help illustrate the use of multiple intentions.

Subscripts on trace events indicate the step in the trace.

$$
\begin{aligned}
&crei((\texttt{percept}, +a), 1)_1\\
&seli(1)_2\\
&addb(a, 1)_3\\
&crei((+a, \epsilon), 2)_4\\
&crei((\texttt{percept}, +b), 3)_5
\end{aligned}
$$

Steps 1–5 in the trace represent two rounds of the reasoning cycle. a is perceived in step 1 and creates an intention (intention 1). Intention 1 is selected (step 2). a is added as a belief and a new intention (intention 2) is created (steps 3 and 4). At the end of the round b is perceived and this creates intention 3 (step 5).

In the next cycle intention 2 is selected. There is no plan for responding to the belief a and so nothing else happens. We get a single addition to the trace: $seli(2)_6$ (intention 1 is empty and is removed. This isn't recorded in the trace).

In the fourth cycle the following steps are added to the trace:

$$
\begin{aligned}
&seli(3)_7\\
&addb(b, 3)_8\\
&crei((+b, \epsilon), 4)_9\\
&crei((\texttt{percept}, +c), 5)_{10}
\end{aligned}
$$

Intention 3 is selected (step 7); b is added to the belief base (step 8); a new intention is created recording the fact (step 9) and; finally, c is perceived (step 10).

In the fifth cycle the following steps are added to the trace:

$$\begin{array}{l} seli(4)_{11} \\ bel((+b, +d; do(e)), a, \emptyset, 4)_{12} \\ selp((+b, +d; do(e)), 4)_{13} \\ addb(d, 4)_{14} \\ crei((+d, \epsilon), 6)_{15} \end{array}$$

Intention 4 is selected (step 11). This triggers the plan and the trace records that the plan's guard, a, holds (step 12) and that the plan has been selected (step 13). The first deed in the plan is executed (d is added to the belief base) (step 14) and this creates intention 6 (step 15).

$$\begin{array}{l} seli(5)_{16} \\ addb(c, 5)_{17} \\ crei((+c, \epsilon), 7)_{18} \end{array} \tag{10}$$

The sixth cycle processes the perception, c.

$$\begin{array}{l} seli(4)_{19} \\ act(e, 4)_{20} \end{array} \tag{11}$$

The seventh cycle selects intention 4 again and this time executes e.

$$\begin{array}{l} seli(6)_{21} \\ seli(7)_{22} \end{array} \tag{12}$$

Finally intentions 6 and 7 are selected in turn. There is no processing to do in relation to them and they are removed.

The agent now has no intentions and execution stops until something new is perceived.

3.3 From Traces to Explanations: Why-Questions in GWENDOLEN

For answering a why-question, a trace is mapped to a chain of reasons (i.e., an explanation). Reasons thus represent a selection of directly connected trace events that might span over large parts of the trace. For example, the trace event of adopting a goal can be directly connected to the event of evaluating the guard of the plan whose body contained that goal, in between evaluating the guard and actually adopting the goal many other trace events could occur (e.g., the evaluations of guards for other plans).

In order to generate explanations we need to link each of the traced events to a local explanation as outlined particularly in [31] but also implied in [15]. To do this the explanation had to be grounded in the specific language, GWENDOLEN, under consideration but nevertheless could be fitted into a general framework.

We consider some event e occurring at step, N, in a trace t and assume, following [31], the existence of a language specific function *why* such that $why(e_N, t)$

returns some representation of an explanation. In our case we represent an explanation as a tree where each node represents a trace event and its children represent previous trace events that explain this one. This tree structures a subset of the trace events that appear in the trace.

Our focus on end users, however, means that explanations should have a default cut-off point and are not unwrapped further unless requested by the user. So we perform further processing on the tree in order to generate our explanations.

Figure 2 shows the algorithm for constructing an explanation tree for GWENDOLEN and Fig. 3 shows the algorithm for converting the tree into a text based explanation.

$why(e_N, \tau)$ can be read as "why did e occur at step N in trace τ" where e is one of our traced events. We can also ask why some formula is believed at step N, $why(b_N, \tau)$, and why some formula is a goal at step N, $why(!g_N, \tau)$.

Definition 1. *An* explanation tree *is a tree structure*

$$t ::= \mathbf{n_e}(e_N, [t, \ldots, t]) \mid \mathbf{n_e}(b_N, [t, \ldots, t]) \mid \mathbf{n_e}(!g_N, [t, \ldots, t]) \mid \mathbf{l_e}(e_N)$$

where e is a trace event, b is a belief formula, g is a goal formula and N is a step in a trace.

Leaves, $\mathbf{l_e}(e_N)$, can be considered as events that require no explanation (such as perceptions) or are to be expanded by a further "top level" question (e.g., "why did you believe that") and nodes, $\mathbf{n_e}(e_n, l)$ are an explanation for trace event e. In selecting the trace events to form part of the explanation tree we typically move backwards along the trace from N looking for an event of some particular kind. We introduce the notation $\uparrow_{N,\tau}\mathcal{S}$ to represent this process where $\mathcal{S}$ is a set of event specifications. An event specification is either an event expression (with capital letters used Prolog-style to indicate variables to be instantiated when a matching event is found) or an event with some side condition – e.g., the event specification $selp((E, D), k) \mid e \in D$ matches a select plan event in intention, k, where e appears in the set of deeds, D, of the plan.

In Fig. 2 we see in equation (13) that an action is taken because some plan was selected that included the action in its deeds. An explanation tree node is constructed with one child – an explanation for why that plan was selected. Note that we need the select plan events to have operated on the same intention (and GWENDOLEN's intention selection mechanism means that other intentions may have been manipulated in between selecting a plan and performing the action) so we track the intention ID number, k, to ensure we are considering the events occurring in the correct intention.

(14) and (15) ask why something is believed at step N or is a goal at step N. In the first case the explanation is that a belief was added to the belief base at some previous step and, in the second case, that an achieve goal event was added to the top of some intention.

The reason an applicable plan is selected (16) is because the guard g was believed and *either* the BDI event e appeared when a new intention was created

$$why(act(a,k)_N,\tau) = \mathbf{n_e}(act(a,k)_N, [why(\uparrow_{N,\tau}\{selp((E,Ds),k) \text{ s.t. } a \in Ds\},\tau)]) \quad (13)$$

$$why(b_N,\tau) = \mathbf{n_e}(b_N, [why(\uparrow_{N,\tau}\{addb(+b,K)\},\tau)]) \quad (14)$$

$$why(!g_N,\tau) = \mathbf{n_e}(!g_N, [why(\uparrow_{N,\tau}\{add((+!g,\epsilon),K)\},\tau)]) \quad (15)$$

$$why(selp((e,ds),k)_N,\tau) = \mathbf{n_e}(selp((e,ds),k)_N, [\mathbf{l_e}(\uparrow_{N,\tau}\{bel((e,ds),g,\theta,k)\}), why(e_\tau,\tau)])$$
$$\text{where} \quad e_\tau = \uparrow_{N,\tau}\{crei((E,\epsilon),k), add((E,\epsilon),k), addb(B,k)\} \quad (16)$$

$$why(addb(+b,k)_N,\tau) = \mathbf{n_e}(addb(+b,k)_N, [why(e_\tau,\tau)])$$
$$\text{where} \quad e_\tau = \uparrow_{N,\tau}\{crei((E,+b),k), selp((E,D),k) \mid +b \in D\} \quad (17)$$

$$why(add((+!g,\epsilon),k)_N,\tau) = \mathbf{n_e}(add((+!g,\epsilon),k)_N, [why(e_\tau,\tau)])$$
$$\text{where} \quad e_\tau = \uparrow_{N,\tau}\{crei((E,+!g),k), selp((E,D),k) \mid +!g \in D\} \quad (18)$$

$$why(crei((a,b),k)_N,\tau) = \begin{cases} \mathbf{l_e}(crei((a,b),k)_N) & a = \texttt{start} \vee \\ & a = \texttt{percept} \\ \mathbf{n_e}(crei((a,b),k)_N, [why(e_\tau,\tau)]) & \text{b} = \epsilon \end{cases}$$
$$\text{where} \quad e_\tau = \uparrow_{N,\tau}\{selp((E,D),K) \mid e \in D, crei((E,e),K)\} \quad (19)$$

Fig. 2. Generating explanation trees

(as happens when beliefs are posted in GWENDOLEN) *or* e was posted to the top of an intention (as happens when goals are posted in GWENDOLEN). We construct a node for the select plan event with two children, the guard event created when the plan's guard was evaluated (for which we do not generate an explanation, though one can be produced if the user wishes) and an explanation for the created intention or posted goal.

(17) and (18) ask why either a belief was added to the belief base or a goal was posted to the top of an intention, this is either because an intention was created to perform that deed (for instance in the case of an initial goal) or a plan was selected previously in the intention which included the posting of the belief or goal as a deed.

There are four reasons why an intention (a, b) may have been created (19): if a is `start` then b was an initial belief or goal, if a is `percept` then b is something perceived. In all other cases $b = \epsilon$ and a is the addition of a belief, and the intention was created either as part of processing an initial belief or percept (i.e., another create-intention event though this time with a on the left hand side) or because a plan was selected which included posting a new belief in its deed stack.

If we look at our running example then if we ask for an explanation for why e was performed in state 20. We get the following explanation tree:

$$\mathbf{n_e}(act(e,4)_{20}, [\mathbf{n_e}(selp((+b,+d;e),4)_{13}, [\mathbf{l_e}(bel((+b,+d;e),a,\emptyset,4)), \mathbf{n_e}(crei((+b,\epsilon),4)_9, [\mathbf{n_e}(addb(b,3)_8, [\mathbf{l_e}(crei((\texttt{percept},+b),3)_5)])])])]) \quad (20)$$

Once we have successfully generated an explanation tree we need to process it into an explanation for presentation to end users. This involves deciding how far down the branches of the tree to progress as part of explanation generation. The algorithm for this is shown in Fig. 3.

In our presentation of explanations we introduce a function *describe*(e) where e is a trace event, a belief formula or a goal formula. We don't present *describe*(e) in full here since it may be application specific. In brief however, we first introduce a *predicate dictionary* for each application that provides a mapping of predicates to strings (e.g., 'state(X)' to 'the robot is in state X'). Second, an internal translation of specific programming symbols is used (e.g., '+!' to 'added the goal').

Key features of the explanation algorithm are that where an explanation involves the selection of a plan we do not always explain why the plan was selected, theoretically leaving the user to expand the explanation if they choose[6]. Secondly explanations never refer to manipulation of intentions (we consider these to be low level details of little interest to most users) so where new intentions are created we do not mention the fact just recursing through to the reason the intention was created (generally the perception of a belief of the posting of an initial goal).

Returning to our running example we generate the following explanation from our explanation tree:

e `was executed because` $+b : \{a\} \leftarrow +d; e$ `was selected in state 13 because` a `was believed and` b `was added in state 8 because` b `was perceived in state 5`.

Our framework is similar in conception and to that in [31] but our focus on end users has caused us to introduce the two step process of buildling an explanation tree structure and then using the *explain* and *describe* functions to present an explanation. That said the actual trace events identified as important to the explanation are generally in agreement with those identified by Winikoff, given our extension to multiple intentions.

Winikoff [31] also treats a number of other trace events—for instance where some deed is not the first to be executed in the body of a plan, then part of the explanation for its execution includes that the previous deeds were successful. We have taken the view that end users will not generally consider "and the parts

[6] It should be noted that our implementation does not yet enable such expansion of explanations.

$$explain(\mathbf{n_e}(act(a,k)_N,[e]) = describe(a)\ \texttt{was executed because}\ explain(e) \quad (21)$$

$$explain(\mathbf{n_e}(b_N,[e])) = describe(b)\ \texttt{was believed in state}\ N\ \texttt{because}\ explain(e) \quad (22)$$

$$explain(\mathbf{n_e}(!g_N,[e])) = describe(g)\ \texttt{was a goal in state}\ N\ \texttt{because}\ explain(e) \quad (23)$$

$$explain(\mathbf{n_e}(selp((e,ds),k)_N,[\mathbf{l_e}(g'_N),e])) = describe((e,ds))\ \texttt{was selected in state}\ N\ \texttt{because}\ describe(g)\ \texttt{and}\ explain(e) \quad (24)$$

$$explain(\mathbf{n_e}(add((+b,\epsilon),k)_N,[crei((e,ds),k)_{N'}]) = describe(+b)\ \texttt{was added in state}\ N\ \texttt{because}\ explain(crei((e,ds),k)_{N'})) \quad (25)$$

$$explain(\mathbf{n_e}(add((+b,\epsilon),k)_N,[selp((e,ds),k)'_N]) = describe(+b)\ \texttt{was added in state}\ N\ \texttt{because}\ describe(selp((e,ds),k))\ \texttt{was selected in state}\ N' \quad (26)$$

$$explain(\mathbf{n_e}(addb((+b,\epsilon),k)_N,[]) = \texttt{the belief}\ describe(b)\ \texttt{was added upon starting the agent} \quad (27)$$

$$explain(\mathbf{n_e}(add((+!g,\epsilon),k)_N,[crei((e,ds),k)_{N'}]) = describe(+!g)\ \texttt{was posted in state}\ N\ \texttt{because}\ explain(crei((e,ds),k)_{N'})) \quad (28)$$

$$explain(\mathbf{n_e}(add((+!g,\epsilon),k)_N,[selp((e,ds),k)'_N]) = describe(+!g)\ \texttt{was posted in state}\ N\ \texttt{because}\ describe(selp((e,ds),k))\ \texttt{was selected in state}\ N' \quad (29)$$

$$explain(\mathbf{l_e}(crei(\texttt{percept},+b)_N)) = describe(+b)\ \texttt{was perceived in state}\ N \quad (30)$$

$$explain(\mathbf{l_e}(crei(\texttt{start},+!g)_N)) = describe(+!g)\ \texttt{was an initial goal} \quad (31)$$

$$explain(\mathbf{n_e}(crei((e,\epsilon),k)_N),[e]) = explain(e) \quad (32)$$

Fig. 3. Generating explanations from an explanation tree

of the plan before this succeeded" as part of an explanation, though we may well need to incorporate aspects of this when we look at why-not-questions (i.e., something may not have happened because a previous deed failed). In general,

our treatment extends that of Winikoff [31] to multiple intentions but does not yet consider why-not questions.

3.4 Implementation

The AIL is implemented in Java. Therefore we were able to create an abstract class for events and a framework for storing and presenting visualisations based on the work in [17]. We were then able to create specific event types for the events of interest. We extended the visualiser with an interface to allow why-questions to be asked—specifically "why did you perform this action?", "why did you hold this belief?" and "why did you have this goal?".

We then constructed a specific explanation mechanism for the GWENDOLEN language based on the algorithms in Figs. 2 and 3.

We also implemented a pretty printing mechanism which utilised the *describe* mechanism to print out traces for user inspection.

This gave us a flexible and extensible framework for implementing omniscient debugging in order to enable why-questions in AIL languages.

4 Test Examples and Evaluation

Our current implementation is a prototype only so a full evaluation has yet to be undertaken. However, it is possible to present initial results.

4.1 Traces in GWENDOLEN for Tutorial Examples

The AIL comes with an extensive set of examples based on tutorials for the framework, the GWENDOLEN language, and the AJPF model-checker [9]. We used these as an ongoing driver for development of our framework—in particular to help settle on appropriate pretty printing conventions. Figure 4 shows part of a pretty printed version of the event trace for one of these examples as it is constructed[7]. The visualiser for traces is shown in Fig. 5. The trace is read from left to right with specifics of various trace events shown on the left.

Figure 6 shows an example explanation (for why the robot performed the action `lift_rubble`).

4.2 Potential Use Case: Self-certifying Offshore Assets

In order to validate our intuitions about appropriate explanations for end users we investigated a prototype agent program for surveying offshore assets such as oil rigs and wind farms [11,12]. This agent guides an unmanned aircraft that must

[7] It is generally accepted that end users prefer natural language presentations while developers often prefer something more compact so this log presents the events with end users in mind, though it remains much more verbose than is required for an explanation.

```
selected Intention 1: add the goal achieve "the robot is holding rubble".
confirmed Intention 1: add the goal achieve "the robot is holding rubble"
can still be processed.
generated 1 applicable plan(s): continue processing: add the goal achieve
"the robot is holding rubble" for an event.
selected continue processing: add the goal achieve "the robot is holding rubble".
added achieve "the robot is holding rubble" to the agent's goals.
modified intention by posting an event to become Intention 1: respond to the
event added the goal achieve "the robot is holding rubble" which has no plan
yet THEN add the goal achieve "the robot is holding rubble".
selected Intention 1: respond to the event added the goal achieve "the robot is
holding rubble" which has no plan yet THEN add the goal achieve "the robot is
holding rubble".
```

Fig. 4. A pretty printed event trace for a GWENDOLEN program

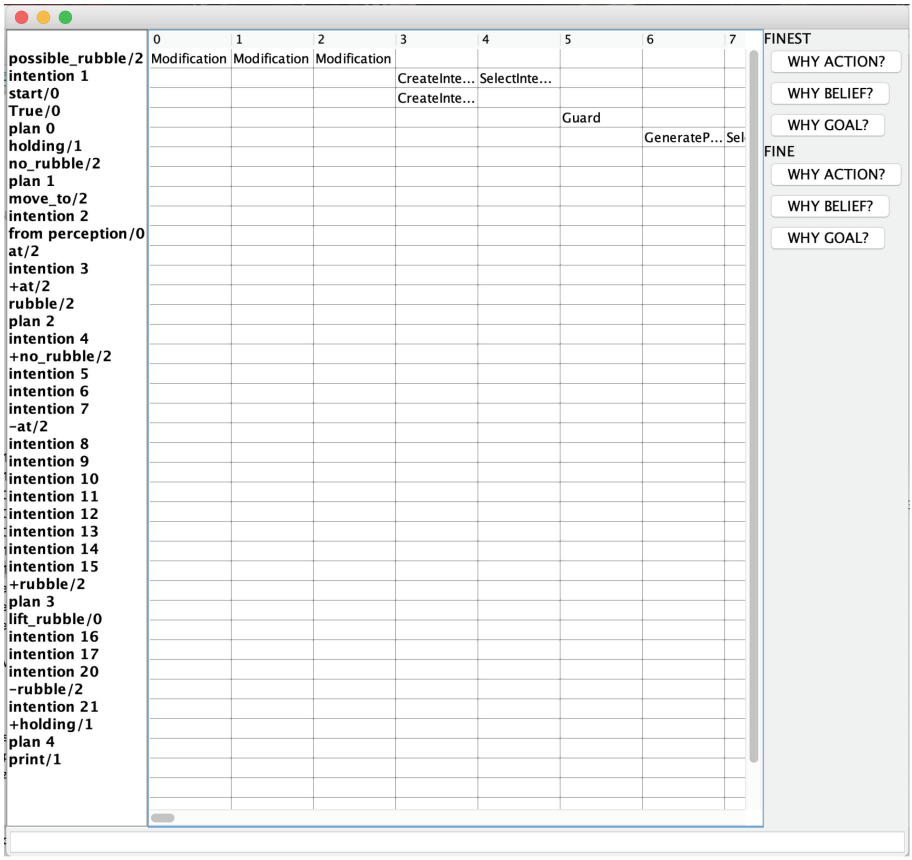

Fig. 5. Trace visualiser

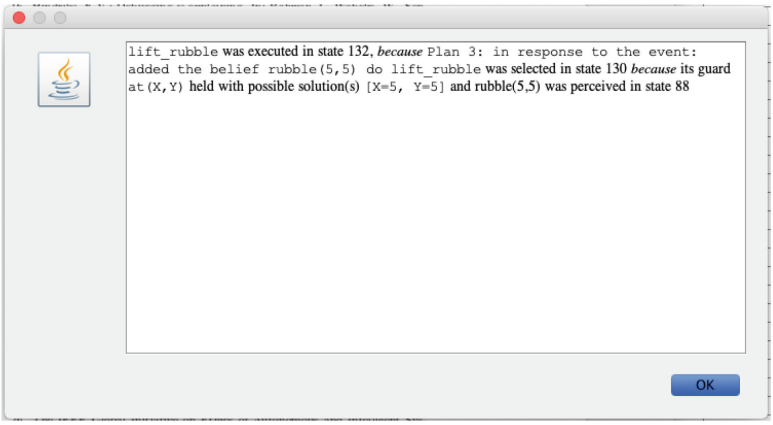

Fig. 6. Generating an explanation

select a suitable path between the legs of an oil rig based on wind speed, wind direction, and perceived tolerance to risks. Guided by the developers, we produced a predicate dictionary for the application which converted the program's internal representation into natural language—e.g., `enactRoute(route2, t2)` becomes `enact route2 with target t2` and so on.

A sample explanation is shown in Fig. 7. These explanations were shown to the developers who confirmed that they provided explanations likely to prove of use to their end users (considered to be experts in unmanned aircraft operation and offshore asset inspection), though obviously further work is needed on the presentation (e.g., performing unifications rather than showing the unifier) and possibly further refining the *explain* algorithm to shorten the initial explanation.

4.3 Traces and Explanations for Other Languages

To evaluate our claim that tracing in the AIL is generic we enabled tracing for another language implemented in the AIL, without any further customisation for the language. The language selected was `pbdi` [4], a reimplementation of a BDI library for Python[8] intended to allow agents written using the library to be verified. We generated an omniscient trace for a simple program in this language (one which stops the operation of a small Pi2Go robot using a command done when the switch on the side of the robot is pressed). A sample trace is shown in Fig. 8.

As can be seen, the lack of language specific pretty printing for plans renders this less readable, but nevertheless a clear trace has been generated of the key events in the execution of the program.

[8] https://github.com/VerifiableAutonomy/BDIPython.

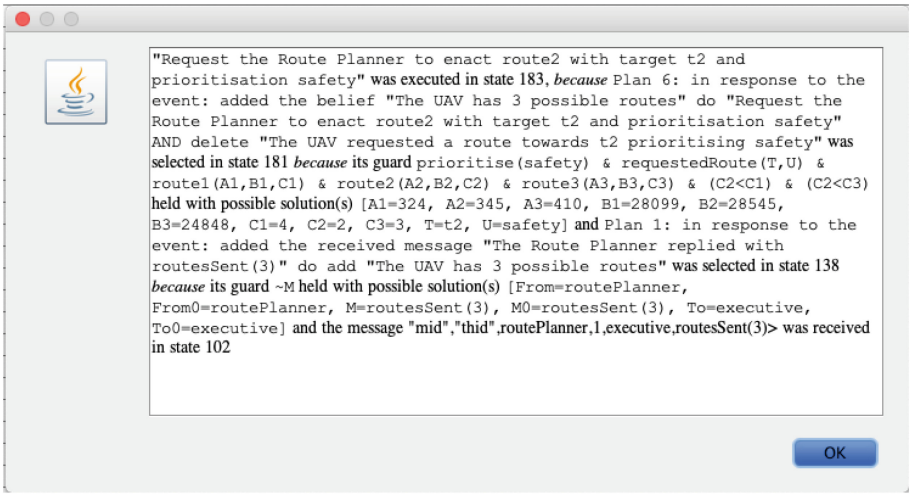

Fig. 7. Explanation of route selection

```
added obstacle_left to the agent's beliefs.
added switch_pressed to the agent's beliefs.
evaluating the guard of 1 :: +!_aAny||True||done||[]
1 :: +!_aAny||switch_pressed||print("Stopping Agent")||[]
 resulted in True.
evaluating the guard of 2 :: +!_aAny||True||print("Obstacle: ",
 agent.sensor_value("obstacle_centre"))||[] resulted in True.
selected 1 :: +!_aAny||True||done||[]
1 :: +!_aAny||switch_pressed||print("Stopping Agent")||[] .
created [empty]::
   *  +state||True||print("Stopping Agent")||[]
      +state||True||done||[] .
selected [empty]::
   *  +state||True||print("Stopping Agent")||[]
      +state||True||done||[] .
Stopping Agent
performed print("Stopping Agent").
performed done.
removed obstacle_left from the agent's beliefs.
removed switch_pressed from the agent's beliefs.
```

Fig. 8. A sample trace for *BDIPython*

5 Conclusion

We sought to combine omniscient debugging and answering why-questions for cognitive agent programs in order to generate explanations for end users. To do this we ported omniscient debugging to the AIL toolkit and thus demonstrated its general applicability beyond the GOAL language for which it was developed.

On top of the traces generated by the omniscient debugger we were able to construct explanations for programs in the Gwendolen language. To do this we extended work by [15] and [31] aimed at answering why-questions for developers. This extension involved adding the capability to handle multiple intentions via the tracking of intention IDs and the use of pretty printing and application specific dictionaries to render explanations into natural language. It would be instructive to perform a full comparison of our algorithm to that in [31] once why-not questions have been tackled.

While this prototype system has yet to be formally evaluated, informal feedback suggests that the end user explanations are appropriate for the intended purpose. A major piece of further work will involve integration of the framework into an application being developed for offshore inspection of oil rigs and wind farms [11,12] and the evaluation of the generated explanations by the application's users. Work is also needed to integrate the answering of why-not-questions into the framework in order to provide *constrastive explanations* as discussed in [18] which argues that why-questions answer counter-factuals.

Work is needed to improve the presentation of traces and explanations and to allow the expansion of explanations if the user wishes to explore further back in a trace. We would also like to investigate the use of tracing/explanation levels analogous to the logging levels used by Java in order to increase the flexibility of the provided explanations allowing users to "drill down" into more detail if the provided explanation does not meet their needs or alternatively to move outward to a presentation similiar to the goal heirarchies used in [13] and [32].

Open Data. The source code for the AIL is available from http://mcapl.sourceforge.net where the work in this paper can be found in the `omniscient` branch of the git repository. The specific examples discussed in the paper can be found in the University of Liverpool Data Catalgue DOI: https://doi.org/10.17638/datacat.liverpool.ac.uk/751

Acknowledgments. This research was partially funded by EPSRC grants Verifiable Autonomy (EP/LO24845/1) and the Offshore Robotics for Certification of Assets (EP/RO26173) Robotics and Artificial Intelligence Hub.

References

1. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley, Hoboken (2007)

2. Bordini, R.H., Dastani, M., Dix, J., El Fallah-Seghrouchni, A. (eds.): Multi-Agent Programming: Languages, Platforms and Applications. Springer, Heidelberg (2005). https://doi.org/10.1007/b137449
3. Bratman, M.E.: Intentions, Plans, and Practical Reason. Harvard University Press, Cambridge (1987)
4. Bremner, P., Dennis, L.A., Fisher, M., Winfiled, A.F.: On proactive, transparent and verifiable ethical reasoning for robots. In: Proceedings of the IEEE special issue on Machine Ethics: The Design and Governance of Ethical AI and Autonomous Systems (2019, to appear)
5. Charisi, V., et al.: Towards moral autonomous systems. CoRR abs/1703.04741 (2017). http://arxiv.org/abs/1703.04741
6. Dastani, M., van Riemsdijk, M.B., Meyer, J.J.C.: Programming multi-agent systems in 3APL. In: [2], chap. 2, pp. 39–67
7. Dennis, L., Fisher, M., Webster, M., Bordini, R.: Model checking agent programming languages. Autom. Softw. Eng. **19**, 1–59 (2011). https://doi.org/10.1007/s10515-011-0088-x
8. Dennis, L.A.: Gwendolen semantics: 2017. Technical report ULCS-17-001, University of Liverpool, Department of Computer Science (2017)
9. Dennis, L.A.: The MCAPL framework including the agent infrastructure layer and agent java pathfinder. J. Open Source Softw. **3**(24), 617 (2018)
10. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A common semantic basis for BDI languages. In: Dastani, M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 124–139. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79043-3_8
11. Dinmohammadi, F., et al.: Certification of safe and trusted robotic inspection of assets. In: 2018 Prognostics and System Health Management Conference (PHM-Chongqing), pp. 276–284, October 2018
12. Fisher, M., et al.: Verifiable self-certifying autonomous systems. In: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 341–348, October 2018
13. Harbers, M.: Explaining agent behaviour in virtual training. Ph.D. thesis, SIKS Dissertation Series (2011). no. 2011–35
14. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming, pp. 119–157. Springer, Boston (2009). https://doi.org/10.1007/978-0-387-89299-3_4
15. Hindriks, K.V.: Debugging is explaining. In: Rahwan, I., Wobcke, W., Sen, S., Sugawara, T. (eds.) PRIMA 2012. LNCS (LNAI), vol. 7455, pp. 31–45. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32729-2_3
16. Ko, A.J., Myers, B.A.: Extracting and answering why and why not questions about Java program output. ACM Trans. Softw. Eng. Methodol. **20**(2), 4:1–4:36 (2010)
17. Koeman, V.J., Hindriks, K.V., Jonker, C.M.: Omniscient debugging for cognitive agent programs. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI 2017, pp. 265–272. AAAI Press (2017)
18. Miller, T.: Explanation in artificial intelligence: insights from the social sciences. Artif. Intell. **267**, 1–38 (2017)
19. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: a BDI reasoning engine. In: [2], pp. 149–174
20. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In: Proceedings of 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR&R), pp. 473–484. Morgan Kaufmann (1991)

21. Rao, A.S., Georgeff, M.P.: An abstract architecture for rational agents. In: Proceedings of International Conference on Knowledge Representation and Reasoning (KR&R), pp. 439–449. Morgan Kaufmann (1992)
22. Rao, A.S., Georgeff, M.P.: BDI agents: from theory to practice. In: Proceedings of 1st International Conference on Multi-Agent Systems (ICMAS), San Francisco, USA, pp. 312–319 (1995)
23. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
24. Sheh, R.K.: "Why did you do that?" Explainable intelligent robots. In: AAAI-17 Workshop on Human-Aware Artificial Intelligence (2017)
25. The IEEE global initiative on ethics of autonomous and intelligent systems: ethically aligned design: a vision for prioritizing human well-being with autonomous and intelligent systems. version 2. Report. IEEE (2017)
26. Turner, J.: Robot Rules: Regulating Artificial Intelligence. Palgrave Macmillan, London (2019)
27. Webster, M., Fisher, M., Cameron, N., Jump, M.: Formal methods for the certification of autonomous unmanned aircraft systems. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 228–242. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24270-0_17
28. Wei, C., Hindriks, K.V.: An agent-based cognitive robot architecture. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS (LNAI), vol. 7837, pp. 54–71. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38700-5_4
29. Winikoff, M., Cranefield, S.: On the testability of BDI agent systems. J. Artif. Intell. Res. **51**, 71–131 (2015)
30. Winikoff, M.: BDI agent testability revisited. Auton. Agents Multi-agent Syst. **31**(1094), 1094–1132 (2017). https://doi.org/10.1007/s10458-016-9356-2
31. Winikoff, M.: Debugging agent programs with Why? questions. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, pp. 251–259. International Foundation for Autonomous Agents and Multiagent Systems, Richland (2017)
32. Winikoff, M., Dignum, V., Dignum, F.: Why bad coffee? Explaining agent plans with valuings. In: Gallina, B., Skavhaug, A., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2018. LNCS, vol. 11094, pp. 521–534. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99229-7_47
33. Wooldridge, M.: An Introduction to Multiagent Systems. Wiley, Hoboken (2002)
34. Wooldridge, M., Rao, A. (eds.): Foundations of Rational Agency. Applied Logic Series. Kluwer Academic Publishers, Berlin (1999)
35. Wortham, R.H., Theodorou, A.: Robot transparency, trust and utility. Connect. Sci. **29**(3), 24200247 (2017)
36. Ziafati, P., Dastani, M., Meyer, J.-J., van der Torre, L.: Agent programming languages requirements for programming autonomous robots. In: Dastani, M., Hübner, J.F., Logan, B. (eds.) ProMAS 2012. LNCS (LNAI), vol. 7837, pp. 35–53. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38700-5_3